Counterfactually Reasoning About Security

Manuel Peralta
Louisiana State University,
Baton Rouge, LA 70803,
U.S.A.
mperal4@lsu.edu

Supratik Mukhopadhyay
Louisiana State University,
Baton Rouge, LA 70803,
U.S.A.
supratik@csc.lsu.edu

Ramesh Bharadwaj
Naval Research Laboratory,
Washington, DC
ramesh@itd.nrl.navy.mil

ABSTRACT

In this paper, we provide the background to counterfactual logic and give very general suggestions on how we could employ this logic to help us reason about security policies. It seems very appropriate to use this kind of logic to anticipate a change that will compromise the security concerns of a given system before actually applying the changes.

Categories and Subject Descriptors

F.3.1 [**Logics and Meaning of Programs**]: Specifying and Verifying and Reasoning about Programs—*logic of programs*; D.4.6 [**Operating Systems**]: Security and Protection—*access controls*

General Terms

Formal Verification, Security

Keywords

formal verification; proof theory; security models; software engineering

1. INTRODUCTION

In the realm of software security, changes regarding security policies are pervasive. It is in this constant changing environment where a system's security becomes compromised. In practice, security policies are changed and, in the worst case, any defect or undesired effect is usually found after the fact and often too late. Requiring that the security policies remain unchanged is out of the question and blatantly unrealistic. Therefore, we are in need of mechanisms that enable us to formalize the security policies, the changes regarding security policies and the future effect of said changes.

In this paper we present a framework for *what-if* analysis of security policies based on Lewis' theory of counterfactuals [7]. The framework can be used to statically perform change-impact analysis for access control matrices. It enables us to verify assertions about a changed version of an access control matrix without actually incorporating the changes. We present a logical calculus that precisely characterizes potential structural modifications to source code and their impact on the program's behavior.

2. RELATED WORK

The theory of counterfactuals allows us to reason about hypothetical situations. It has been used in Philosophy and Political Science [9] for decision making in a hypothetical environment. In Physics, it has been used for reasoning about measurements in quantum mechanics [12], [8]. The main idea exposed by Fisler et al. in [2] is to gain knowledge regarding the effects of changing access control policies before actually making such changes. The work of Fisler et al. is similar to the one presented in this paper as it tries to find the effects of a change *a priori.* The work of Chockler et al. in [1] employs counterfactual reasoning also in the context of model checking. In this instance, the authors emphasize their work in coverage issues. They use counterfactual reasoning to enhance the coverage information. This work differs from ours since they use counterfactual logic to explore alternative scenarios whereas we explore a single alternative version given an initial version.

Also, in [4], Groce et al. use *counterfactual theory* to detect failures, isolate errors and aid in repairing modifications of program code in the context of model checking. They construct a model of program executions and establish a metric among them. This metric lets them analyze faulty executions by examining those which lie at some distance from a given faulty execution. The work of [4] relates to ours in the sense that the authors define a notion of *distance* among possible execution traces in the same sense we implicitly define the number of transformation steps between program versions. In [4], however, the authors go beyond just defining a notion of proximity and actually define, given a system execution trace, the set of neighboring traces whereas our work only characterizes a single future version. The work of Ren et al. in [11] exposes a tool (i.e. *Chianti*) that analyzes two different versions of a given program and a set of test cases for such program and determines which tests are affected due to the changes that lead from one version to another. Furthermore, for each affected test, the tool determines a set of method-level changes that most probably affected the test. Our work differs from the work in [11] in the sense that we do not require a second version and a set of test cases. Our approach just needs the changes to be expressed in our logical calculus. The approach given by Guo et al. in [5] exhibits a method by which change impact analysis is modeled and verified in a distributed setting. Their model is

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **NOV 2011**	2. REPORT TYPE **N/A**	3. DATES COVERED **-**
4. TITLE AND SUBTITLE **Counterfactually Reasoning About Security**		5a. CONTRACT NUMBER
		5b. GRANT NUMBER
		5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)		5d. PROJECT NUMBER
		5e. TASK NUMBER
		5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Louisiana State University, Baton Rouge, LA 70803, U.S.A.**		8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)		10. SPONSOR/MONITOR'S ACRONYM(S)
		11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release, distribution unlimited

13. SUPPLEMENTARY NOTES
See also ADA553912. International Conference on Security of Information and Networks (4th) (SIN 2011) Held in Sydney, Australia on November 14-19, 2011. Approved for public release; U.S. Government or Federal Purpose Rights License.

14. ABSTRACT
In this paper, we provide the background to counterfactual logic and give very general suggestions on how we could employ this logic to help us reason about security policies. It seems very appropriate to use this kind of logic to anticipate a change that will compromise the security concerns of a given system before actually applying the changes.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**SAR**	**4**	

in essence a network of state machines that communicate either via shared variables or queues. Changes are modeled as adding and/or deleting transitions from the composite state machine that represents the distributed system. The work in [5] is similar to ours in the context of two aspects: 1) the authors are formally representing change in a system, however, their approach targets distributed computations and is based on model checking whereas our approach targets sequential computation and it is based on theorem proving; 2) this approach prunes the global state space by using *partial order reduction* in order to infer the valid transitions when a change occurs; our approach deals with change at the source code level and the validity of the change is inferred by our logical calculus.

In [13], Subramaniam et al. enhance the approach shown in [5]. The changes are still represented by adding and/or deleting transitions of a composite state machine. However, this work addresses the issue of test suite coverage when changes occur. This approach detects the affected tests based on whether or not these include the affected transitions. Using formal verification techniques similar to the ones presented in [5], the authors are able to reduce the total regression test suite by identifying which tests are relevant after a given change. Our approach goes in a different direction by formally characterizing the source code-change and determining if the changes to the current source code version are logically consistent with its properties and the future desired properties.

The work presented in [10] uses symbolic execution to establish whether or not two source code versions are equivalent. In the negative case the proposed approach generates the *deltas* which characterize the input values that induce the behavior difference between the two versions. Our approach, being based on proof-theoretic methods, relies mostly on the syntactic nature of change and hence we consider two versions identical as long as they have equivalent logical characterizations. The latter also means that we are only interested on cases where our two versions (the actual and the potential version) are logically different.

3. COUNTERFACTUAL THEORY

The logic of counterfactuals helps us reason about assertions that are not a matter of fact. In [7] Lewis provided a sound and complete proof system and proved its decidability. Based on the logic of counterfactuals we derive a logical calculus that allows us to assert properties that would hold for a future version of a given program and verify that these would indeed hold if the changes needed to obtain that version were actually implemented.

3.1 The Language of Counterfactual Theory

In coherence with [7] we will briefly introduce the language regarding the logic of counterfactuals below (p_i denotes a propositional variable):

$$\phi ::= p_i | \neg\phi | \phi \wedge \phi | \phi \vee \phi | \phi \rightarrow \phi | \phi \,\Box\!\!\rightarrow \phi$$

The counterfactual sentence $\phi \,\Box\!\!\rightarrow \psi$ should be read as: **if it had been the case that ϕ, it would have been the case that ψ.** Thus, if we had an assertion whose antecedent ranged over the properties of some given access control matrix and also the changes needed to produce a new version and whose consequent ranged over the properties that a new version would have, then we could use counterfactual logic to code such an assertion.

3.2 Formal Representation of A Security Model

In this section we will define a simple variation of the Access Control Matrix (ACM) model. This model was first introduced in [6] and [3]. We have chosen this model due to its simplicity and readily intuitive nature and widespread use as it is stated in [6]. First of all, we will define three sets: S, O and A which are respectively the set of : subjects, objects and actions. The set of subjects contains the active entities on the system (i.e. users, computer systems, etc.); the set of objects denotes the set of entities over which subject are allowed or denied a certain action. The set of actions denotes those tasks which a subject can perform on a given object. Hence:

S	$=$	$\{s_i\}_{i\in I}$	The set of subjects
O	$=$	$\{o_j\}_{j\in J}$	The set of objects
A	$=$	$\{Read, Write\}$	The set of actions

Thus, we can now formally define an *access control matrix* as a function $M : S \times O \rightarrow 2^A$ which takes an ordered pair composed of a subject and an object and assigns to them a subset of the possible set of actions. Moreover, in this instance we will have to work with M's *intentional* or set representation and thence:

$$M \triangleq \{(s_i, o_j, \alpha_k)\} \text{ where } \alpha_k \in 2^A$$

3.2.1 Encoding Change in the ACM model

Let $M = \{(s_i, o_j, \alpha_k)_{i,j,k\in\mathbb{N}}\}$ be the current version of the access control matrix. We can encode the state of any ACM by using the following formula:

$$\Psi_M \triangleq \bigwedge_{i=1}^{n} \bigwedge_{j=1}^{m} \bigwedge_{k=1}^{p} (s_i, o_j, \alpha_k) \in M$$

We will simplify the latter expression using the following notation:

$$\Psi_M \triangleq \bigwedge_{i,j,k} (s_i, o_j, \alpha_k)$$

For the sake of simplicity we will define a change to the ACM as a change to the members of the action-set of a given triple. Hence, a change may be represented as:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k)$$

It can be readily inferred that $\Rightarrow^c$ is a three-place-relation over $S \times O \times A$. Furthermore, let $\mathcal{M}$ denote the class of all possible ACM versions. Therefore, $\Rightarrow^c$ can be thought of as a binary relation over $\mathcal{M}$ and

$$M \Rightarrow^c M' \text{ iff } M' \triangleq M[\alpha_k/\alpha'_k]$$

Moreover we define $\Rightarrow^c$ to be the smallest relation such that the following holds:

$$(s_i, o_j, \alpha_k) \Rightarrow^c (s_i, o_j, \alpha'_k) \text{ iff:}$$

1. $\alpha'_k \neq \emptyset$ when $\alpha_k \neq \emptyset$
2. $\alpha'_k \neq A$ when $\alpha_k = A$

3.2.2 *Undesirable Configurations*

In any system, there is a set of undesirable states. These states may be very possibly members of the entire set of possible states. One of the fundamental purposes of any security mechanisms is to guarantee that for any possible transition (that originates in a safe/legal state) the target state will not be an illegal/undesirable state. In this instance an *undesired state* will be denoted by a given configuration/triple of subject, object and action. Therefore, let $U \subseteq S \times O \times A$ be the set of illegal configurations and let τ_u range over this set.

We want to avoid allowing a configuration change in which we enable an undesirable triple be part of the new version of the ACM. Hence, we want to avoid the following:

$$M \Rightarrow^c M' \text{ where } \tau_u \in M'$$

3.2.3 *Secure Counterfactual Change*

Our objective is to enable *counterfactual logic* to let us decide whether or not a change to the current version of the ACM implies that at least one illegal triple is part of the future resulting version. Thence our secure counterfactual implication can be expressed as:

$$[\bigwedge_{i,j,k \in \mathbb{N}} (s_i, o_j, \alpha_k)] \wedge (s_0, o_0, \alpha_0)[\alpha_0/\alpha_0'] \;\Box\!\!\rightarrow (\tau_u \notin M')$$

3.3 Kripke Versioning Model

In [7] the author provides the semantics of his counterfactual propositional logic using a multiple-world interpretation. In that same manner we have chosen to interpret our access control matrix transformation. In our case, each ACM version will represent a world. In the following definition, we take the liberty of writing $t_i \leftarrow t_j$ to denote that the tuple t_i was swapped by tuple t_j. Below, we provide a formal interpretation based on a Kripke model.

Definition 3.1 (Kripke Version Model). *A Kripke Version Model $\mathcal{R}$ is a triple $\langle \mathcal{M}, \Rightarrow, M_0 \rangle$ where:*

1. *$\mathcal{M} = \{M_k\}_{k \in \mathbb{N}}$ is the set of all access control matrix versions (ACM states).*
2. *M_0 is the initial access control matrix.*
3. *$\Rightarrow \subseteq \mathcal{M} \times \mathcal{M}$ is a binary relation defined the set of all possible ACM versions. Where $\Rightarrow$ is the smallest relation such that the following properties hold:*
   (a) *$t_i \leftarrow t_i$: tuple s_i is left unchanged. This stands for the do nothing transformation.*
   (b) *$t_i \leftarrow t_j$: tuple t_j replaces statement t_i, where $t_j \in M_k$. We usually call this primitive transformation, a swap.*
   (c) *$t_i \leftarrow t_j$: statement t_j replaces statement t_i, where $t_j \notin M_k$. Thus, s_k is a new statement.*
   (d) *$(\forall i) t_i \in M_k$ can be changed only once.*

Furthermore, we assume that the relation $\Rightarrow$ complies with the properties of *reflexivity*, *symmetry*, and *transitivity*. Below, we justify each property based on the latter definition of $\Rightarrow$:

1. **Reflexivity:** For any ACM $M_i \in \mathcal{M}$, it is obvious that the *do-nothing transformation* will yield that any ACM can be transformed into itself. Therefore, $M_i \Rightarrow M_i$ given that for all $s_j \in M_i$, $M_i = M_i[s_j/s_j]$
2. **Symmetry:** For any ACMs $M_i, M_j \in \mathcal{M}$ any of the above transformations can be reversed and therefore, $M_i \Rightarrow M_j$ implies $M_j \Rightarrow M_i$.
3. **Transitivity:** For any ACMs $M_i, M_j, M_k \in \mathcal{M}$, applying two or more transformations to a program will yield intermediate versions; this is equivalent to transforming the initial version by composing the transformations into one. Thus, $M_i \Rightarrow M_j$ and $M_j \Rightarrow M_k$ imply that $M_i \Rightarrow^+ M_k$. Where $\Rightarrow^+$ denotes $\Rightarrow \circ \Rightarrow^{n-1}$ and $n > 1$.

3.4 Interpreting the Counterfactual Implication

As it was stated earlier, the purpose of our model is to help interpret assertions in the language of counterfactual logic. Let M_0 denote our given initial ACM version. Also, let us assume we had a counterfactual assertion, namely $\phi \;\Box\!\!\rightarrow \psi$ in which:

- ϕ stands for assertions regarding M_0 and some transformation $t_i \leftarrow t_j$ that implies that $M_1 = M_0[t_i/t_j]$
- ψ stands for assertions regarding M_1

Thus, following the model-theoretic interpretation proposed by Lewis in [7], our version of the counterfactual implication is interpreted as:

$$\mathcal{R} \models \phi \;\Box\!\!\rightarrow \psi \;. \qquad (1)$$

Where $\mathcal{R}$ denotes our previously defined *Kripke Versioning Model.* Moreover, letting α_i, β_i denote propositional statements about the structure of M_i and M_j respectively, then, we can state that:

$$\begin{aligned} \phi &\triangleq (\bigwedge_{i=1}^{n} \alpha_i) \wedge (M_i \Rightarrow^+ M_j) \\ \psi &\triangleq \bigwedge_{j=1}^{m} \beta_j \end{aligned}$$

Where $\Rightarrow^+$ denotes the positive/transtive closure for the relation $\Rightarrow$. Furthermore, given an initial ACM version, namely M_0, we produce several versions by applying one or more transformations to it. In the context of a counterfactual assertion, the current version's structure and the changes applied to it (in order to produce a new version) imply properties possessed the new version and hence:

$$\begin{aligned} \mathcal{R} \models \phi \;\Box\!\!\rightarrow \psi \;\triangleq\; & (\exists_{\min} k \in \mathbb{N})(\textstyle\bigwedge_{i=1}^{n} \alpha_i) \\ & \wedge (M_0 \Rightarrow^k M') \rightarrow (\textstyle\bigwedge_{j=1}^{m} \beta_j) \;. \end{aligned}$$

The latter should be interpreted as there exists a minimal number of transformation steps such that given the properties of our initial ACM M_0 (namely, $\bigwedge_{i=1}^{n} \alpha_i$) and the transformation between the two program versions implies the desired properties of the future ACM version (namely, $\bigwedge_{j=1}^{m} \beta_j$).

4. APPLICATIONS OF COUNTERFACTUAL THEORY TO SECURITY

Each change to the access control matrix modifies the state of the security system. Hence, each change reflects a change in the set of valid policies. It seems very promising to use counterfactual logic to 1) encode the changes to the ACM, 2) express the undesirable state-tuples, and 3) assert whether or not the changes *counterfactually* imply the undesirable tuples are part of the future state of the ACM. Given all the risks involved in changing security policies, it would be nice to foresee their effect before incorporating them into production systems.

5. CONCLUSION AND FUTURE WORK

We have introduced a logical calculus based on Lewis' theory of counterfactuals. Additionally we have shown that if we know how to unambiguously characterize the transformation from the initial ACM state to the future desired ACM state, the conjunction between the structural properties of the initial ACM version and the predicates that characterize the transformation, imply the desired future ACM state's properties.

This position paper has presented a powerful and promising suggestion which consists of jointly using a perhaps modified version of the ACM model and our counterfactual logical calculus. The latter mix would enable practitioners verify *a-priory* the effects of a change to the ACM without actually applying the change to production systems. Although it is widely known that the question of whether or not a given security model enforces a given policy is a non-decidable problem, we are confident that our simplified ACM model and our counterfactual logic will be helpful to enough non-trivial applications.

6. REFERENCES

[1] H. Chockler, J. Y. Halpern, and O. Kupferman. What causes a system to satisfy a specification? *ACM Trans. Comput. Logic*, 9(3):1–26, June 2008.

[2] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 196–205, New York, NY, USA, 2005. ACM.

[3] G. S. Graham and P. J. Denning. Protection: principles and practice. In *Proceedings of the May 16-18, 1972, spring joint computer conference*, AFIPS '72 (Spring), pages 417–429, New York, NY, USA, 1972. ACM.

[4] A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error explanation with distance metrics. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(3):229–247, June 2006.

[5] B. Guo and M. Subramaniam. Formal change impact analyses of extended finite state machines using a theorem prover. *Software Engineering and Formal Methods, International Conference on*, pages 335–344, 2008.

[6] B. W. Lampson. Protection. *SIGOPS Oper. Syst. Rev.*, 8(1):18–24, Jan. 1974.

[7] D. K. Lewis. *Counterfactuals*, chapter 6, pages 118 – 143. Wiley-Blackwell, 2nd edition, January 2001.

[8] G. Mitchison and R. Jozsa. Counterfactual computation. *Proceedings of the Royal Society of London. Series A: Mathematical, Physical and Engineering Sciences*, 457(2009):1175–1193, May 2001.

[9] J. Pearl. *Causality: Models, Reasoning, and Inference.* Cambridge University Press, March 2000.

[10] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, SIGSOFT '08/FSE-16, pages 226–237, New York, NY, USA, 2008. ACM.

[11] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 39, pages 432–448, New York, NY, USA, October 2004. ACM.

[12] B. Skyrms. Counterfactual definiteness and local causation. *Philosophy of Science*, 49(1):43–50, 1982.

[13] M. Subramaniam, B. Guo, and Z. Pap. Using change impact analysis to select tests for extended finite state machines. *Software Engineering and Formal Methods, International Conference on*, pages 93–102, 2009.